微内核

2023年7月12日 22:04

内核，开发环境。内核应该有什么功能?

宏内核：可以提供强大抽象

- 可移植性
- 隐藏复杂性
- 共享资源，安全性
- 性能好，可以访问各自的数据文件

为什么不?

- 庞大->复杂（相互依赖）->BUG->安全性
- linux让自己通用，但是webserver可能不需要声卡，所以这会导致慢。衍生：每个场景都最快了嘛?
- 受API控制，必须遵守某些控制，这个导致不方便。
- 数据库可能写B树，但是文件系统并不知道，文件系统不能有效利用用户程序的一些特征。
- 可拓展性不好，有时候用户想修改内核，但是不行

微内核（一种通用的设计方法）

- 只提供IPC（Inter-Process Communication）进程间通信
- 只提供进程抽象（address space 和 task）和支持通信的tiny kernel

其他什么都没有，比如驱动，网络协议栈，优化都不多。
不过宏内核也支持IPC。

构建动机

- linux大内核，**不优雅**。
- 小内核安全，更易优化，可能会更快，因为可以扔掉很多功能，设计限制更少，灵活。

【特殊情况下需要正确性（bug free）大内核难以被验证，小是证明安全的关键。】
【但是上面都不是必须的】

- 拆分内核，代码更加模块化。
- 用户态代码一般更容易被修改。
- 更加健壮，一个功能模块崩溃害不到其他地方。
- 你可以在微内核里面运行多个OS!

挑战

- 你需要提供多少系统调用？内核不知道文件系统怎么实现exec？必然需要用户程序支持。
- IPC会有大量通信，要多快才能有竞争力？同时模块化会导致集成的时候难以优化。

例子

- 系统抽象极少（7个，相比linux 350个）
- 代码长度只有linux 1/20
- 只包含基础抽象
  - Task（进程），线程，地址空间，IPC

用IPC转换中断（比如网卡），trap（比如PF），etc（所以IPC系统需要非常非常快。）
线程切换和XV6很类似。
用用户空间代码获取PF，这样就不会弄乱内核。

- IPC设计

早期，类似于XV6的pipe，拥有缓存，异步传输不用等待还好，但是如果需要来回发送者会开始等待，性能下降的厉害。而且出让CPU，发送接收，都需要系统调用。

- 4个系统调用，两个send，两个recv
- 对应8次用户空间内核空间之间的切换，而每一次切换明显都会很慢
- 在recv的时候，需要通过sleep来等待数据出现
- 并且需要至少一次线程调度和context switching来从P1切换到P2

切换时的页表切换，会清空TLB（不绝对）
但这只是一个简单的场景，发送消息并且期待收到回复。

- **同步**（所以可以弃用缓存）send会等待P2调用recv，两者都进入内核空间后，
- P2就相当于从recv中返回，不需要线程调度。
- 一个地址空间，所以可以直接copy到其他线程，不需要经过内核转手
- 消息很小的时候可以直接保存在特定寄存器。
- 非常长消息，只需要提供共享page权限（快于拷贝）
- RPC（这种request 和 response的一种常见特殊情况）专门用一个系统调用减少切换消耗。用sendrecv，发送以一个消息，然后开始等待，节省一半切换开销。（你想要发送的消息，以及你要存放回复消息的位置。所以结合了对应操作。）（这个不会有安全问题吗？）

对于短消息RPC（同样是一个典型的场景）可以提升20倍，今天RPC的快速，让人们更加认同微内核。
关于P2为什么知道自己要调用recv
P2可能是一个server，所以可能一直循环在等待消息。
P1进入到内核，但是却返回到P2，有点奇怪但是非常快。（我想到了线程的切换。）

解决移植性
把linux作为一个对应的task，然后把linux一些直接的修改改为系统调用就行。

- 线程是个问题，L4破坏了linux的线程调度喜欢哪个用哪个（但是linux又有自己私有的内核线程调度）

（发布的时候还没有CPU多核硬件，而且那个时候Linux不支持多CPU）
这样子搞，完全忽视了Linux内部已有的线程调度机制。（相信会被修复。）

第一代微内核很差，有很多问题，为了测试我们精简的内核有没有克服缺点，我们对比了原生linux和跑在一代微内核的机子。
结果是完爆一代u内核，和原版只差5%
我们这个内核，拓展性很好。甚至内存分配（包括二级cache的分配）都可以在user space跑。
大家因为第一代微内核表现太差，就放弃了讨论，大部分人都觉得微内核的抽象太高或者太低。太高的可以向内核加方法，太低的构建内核用类似于硬件架构构建软件。
别人说在特定架构上微内核可以很快，但是在一般机子会损失很多性能。
但是通过我系统的研究，区别于一代从宏内核演进过来的方法，我们二代从头开始设计，目标就是最小化。在实践中，微内核是能用的，而且性能不错。
为了显示系统的拓展性和能实现的性能。

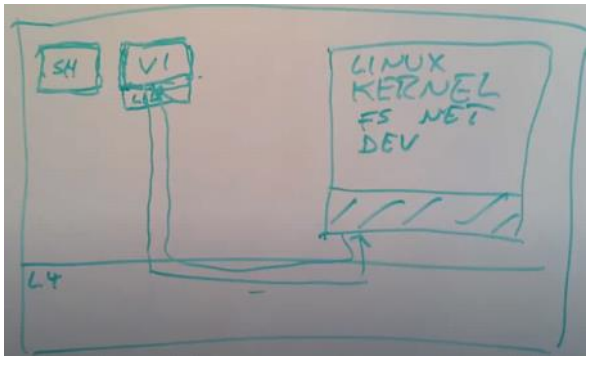
- 原生的管道对比，然后支持了一些基于映射的OS拓展，还有就是在user space实现实时内存分配来提升实时最糟情况的性能

为了看我们自己的抽象是否合理，我们在另一个机子上重新实现了一遍自己的系统。
还在IPC的通信实现上又封装了一层看看高级点的通信抽象是不是比低级的好。

嵌套的地址空间表示。
一开始的地址空间，0实际上是物理内存。
然后后续的地址空间的构建是使用，granting，mapping，unmapping来构建的
大小都是2^n，拥有者是可以把自己的地址空间映射到其他的地址空间里面，不过前提是接收者同意。

map，unmap
这三个操作是安全的，因为这三个东西是在虚拟页上工作的，但不是在物理frame（页框）上进行的，所有只能map和unmap自己有的，

如果P1要完成一次完整的消息发送和接收，那么可以假设有两个buffer，一个用来发送消息，一个用来接收消息。P1会先调用send，send返回之后。之后P1会立即调用recv，recv会等待接收消息的buffer出现数据，所以P1会出让CPU。在一个单CPU的系统中，只有当P1出让了CPU，P2才可以运行。论文中的讨论是基于单CPU系统，所以P1先执行，之后P1不再执行，出让CPU并等待回复消息。这时，P2才会被调度，之后P2调用recv，拷贝消息。之后P2自己再调用send将回复消息追加到buffer，之后P2的send系统调用返回。假设在某个时间，或许因为定时器中断触发导致P2出让CPU，这时P1可以恢复运行，内核发现在接收消息buffer有了一条消息，会返回到用户空间的P1进程。



把linux作为一个对应的task，然后把linux一些直接的修改改为系统调用就行。

- 线程是个问题，L4破坏了linux的线程调度喜欢哪个用哪个（但是linux又有自己私有的内核线程调度）
  （发布的时候还没有CPU多核硬件，而且那个时候Linux不支持多CPU）
  这样子搞，完全忽视了Linux大量且复杂的线程调度机制。（相信会被修复。）

切换页表的成本主要是因为TLB flush
现在一般都是杂交（hybrid）的核。

你应该问自己：通过论文可以学到有关微内核的什么内容呢？

对于我们来说，论文中有很多有趣的有关微内核是如何运行，有关Linux是如何运行的小的知识点，以及你该如何设计这么一个系统。但是论文并没有回答这个问题：微内核是不是一个好的设计？论文只是讨论了微内核是否有足够的性能以值得使用。

论文之所以讨论这个内容的原因是，在论文发表的前5-10年，有一场著名的测试针对一种更早的叫做MACH的微内核。它也运行了与L4类似的结构，但是内部的设计完全不一样。通过测试发现，当按照前一节的架构运行时，MACH明显慢于普通的Unix。这里有很多原因，比如IPC系统并没有如你期望的一样被优化，这样会有更多的用户空间和内核空间的转换，cache-miss等等。有很多原因使得MACH很慢。但是很多人并不关心原因，只是看到了这个测试结果，发现MACH慢于原生的操作系统，并坚信微内核是无可救药的低效，几乎不可能足够快且足够有竞争力。很多人相信应该都使用monolithic kernel。

今天的论文像是对于这种观点的一个反驳，论文中的观点是，你可以构建类似上一节的架构，如果你花费足够的精力去优化性能，你可以获取与原生操作系统相比差不多的性能。因此，你不能只是因为性能就忽视微内核。今天的论文要说明的是，你可以因为其他原因不喜欢微内核，但是你不能使用性能作为拒绝微内核的原因。

达成这一点的一个重要部分是，IPC被优化的快得多了，相应的技术在18.5中提到过。

让时间快进20年，如果之前所说，现在人们实际上在一些嵌入式系统中使用L4，尤其在智能手机里有很多L4实例在运行，它们与Unix并没有兼容性。在一些更通用的场景下，像是工作站和服务器，微内核从来没有真正的流行过，并不是因为这里的设计有什么问题，只是为了能够吸引一些软件，微内核需要做的更好，这样人们才会有动力切换到微内核。对于人们来说很难决定微内核是否足够好，这样才值得让他们经历从现在正在运行的Linux或者其他系统迁移到微内核的所需要的各种麻烦事。所以，微内核从来没有真正流行过，因为它们并没有明显的更好。